08/817549
88 Rec'd PCT/PTO 16 APR 1997

APPARATUS AND METHOD FOR COMPRESSING BINARIZED IMAGES

FIELD OF THE INVENTION

The present invention relates to methods for compressing binarized images, generally.

BACKGROUND OF THE INVENTION

Arithmetic coding is described in:

Witten, I. H et al, "Arithmetic coding for data compression", Computing Practices, Communications of the ACM, Jun 1987, Vol. 30(6); and

"Arithmetic coding and statistical modeling", Dr. Dobb's Journal, Feb. 1991, pp. 16 - 29.

The MR decoding scheme is described in CCITT Recommendation T.4 and T.6 for Groups 3 and 4.

A conventional binarizing technique is described in Foley, J. et al, Computer Graphics: Principles and practice, 2nd Ed., Section 13.1.2, pages 568 - 573.

The disclosures of all of the above publications are hereby incorporated by reference.

SUMMARY OF THE INVENTION

The present invention seeks to provide an improved image manipulation system.

There is thus provided in accordance with a preferred embodiment of the present invention a method for compressing binarized images including receiving a binarized image and generating a first sequence of first code symbols representing the binarized image wherein at least one row of the image is represented in run-length encoded format, and encoding a portion of the first sequence of code symbols using a preliminary encoding scheme, thereby to provide a first portion of a second sequence of code symbols, and, while encoding, accumulating the frequency of at least some of the first code symbols thus far encoded and generating an additional portion of the second sequence using a modified version of the code scheme such that at least one subsequent code symbol in the first sequence with a large accumulated frequency is encoded more compactly in the second portion than at least one subsequent code symbol in the first sequence with a small accumulated frequency.

Further in accordance with a preferred embodiment of the present invention, a modified Huffman coding scheme is employed to generate the first sequence of first code symbols.

In accordance with another preferred embodiment of the present invention, there is provided a method for compressing binarized images including receiving a binarized image and generating a first sequence of first code symbols representing the binarized image including a representation of one row of the binarized image and a representation of differences between at least one subsequent row and at least one previous row, and encoding a

portion of the first sequence of code symbols using a preliminary encoding scheme, thereby to provide a first portion of a second sequence of code symbols, and, while encoding, accumulating the frequency of at least some of the first code symbols thus far encoded and generating an additional portion of the second sequence using a modified version of the code scheme such that at least one subsequent code symbol in the first sequence with a large accumulated frequency is encoded more compactly in the second portion than at least one subsequent code symbol in the first sequence with a small accumulated frequency.

Further in accordance with a preferred embodiment of the present invention, the encoding scheme used to encode the first sequence of code symbols is continually modified such that code symbols in the first sequence with a large accumulated frequency are encoded more compactly in the second portion than subsequent code symbols in the first sequence with a small accumulated frequency.

Still further in accordance with a preferred embodiment of the present invention, a modified-read coding scheme is employed to generate the first sequence of first code symbols.

Further in accordance with a preferred embodiment of the present invention, a modified modified-read coding scheme is employed to generate the first sequence of first code symbols.

Still further in accordance with a preferred embodiment of the present invention, the method also includes binarizing a discrete level image, thereby to provide the binarized image.

Additionally in accordance with a preferred embodiment of the present invention, the method also includes binarizing a continuous level image, thereby to provide the binarized image.

Still further in accordance with a preferred

embodiment of the present invention, arithmetic coding is employed to translate the accumulated frequency of at least some of the first code symbols into second code symbols.

There is also provided, in accordance with a preferred embodiment of the present invention, apparatus for compressing binarized images including a run-length encoder operative to receive a binarized image and to generate a first sequence of first code symbols representing the binarized image wherein at least one row of the image is represented in run-length encoded format, and an adaptive encoder operative to encode a portion of the first sequence of code symbols using a preliminary encoding scheme, thereby to provide a first portion of a second sequence of code symbols, and, while encoding, to accumulate the frequency of at least some of the first code symbols thus far encoded and to generate an additional portion of the second sequence using a modified version of the code scheme such that at least one subsequent code symbol in the first sequence with a large accumulated frequency is encoded more compactly in the second portion than at least one subsequent code symbol in the first sequence with a small accumulated frequency.

There is further provided, in accordance with a preferred embodiment of the present invention, apparatus for compressing binarized images including a binarized image compressor operative to receive a binarized image and to generate a first sequence of first code symbols representing the binarized image, the first sequence including a representation of one row of the binarized image and a representation of differences between at least one subsequent row and at least one previous row, and an adaptive encoder operative to encode a portion of the first sequence of code symbols using a preliminary encoding scheme, thereby to provide a first portion of a second sequence of code symbols, and, while encoding, to

accumulate the frequency of at least some of the first code symbols thus far encoded and to generate an additional portion of the second sequence using a modified version of the code scheme such that at least one subsequent code symbol in the first sequence with a large accumulated frequency is encoded more compactly in the second portion than at least one subsequent code symbol in the first sequence with a small accumulated frequency.

Further in accordance with a preferred embodiment of the present invention, the binarized image compressor employs a modified-read coding scheme to generate the first sequence of first code symbols.

Still further in accordance with a preferred embodiment of the present invention, the binarized image compressor employs a modified modified-read coding scheme to generate the first sequence of first code symbols.

Additionally in accordance with a preferred embodiment of the present invention, the adaptive encoder employs arithmetic coding to translate the accumulated frequency of at least some of the first code symbols into second code symbols.

Still further in accordance with a preferred embodiment of the present invention, the encoding scheme used to encode the first sequence of code symbols is continually modified such that code symbols in the first sequence with a large accumulated frequency are encoded more compactly in the second portion than subsequent code symbols in the first sequence with a small accumulated frequency.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be understood and appreciated from the following detailed description, taken in conjunction with the drawings in which:

Fig. 1 is a simplified block diagram of an image manipulation system constructed and operative in accordance with a preferred embodiment of the present invention, and

Fig. 2 is a simplified flowchart illustrating a preferred mode of operation in which the MR code element frequency accumulation unit of Fig. 1 processes a single MR code element in a sequence.

Attached herewith are the following appendices which aid in the understanding and appreciation of one preferred embodiment of the invention shown and described herein:

Appendix A is a computer listing of a preferred software embodiment of the MR coding, arithmetic coding and MR code element frequency accumulation units of Fig. 1, and

Appendix B is a computer listing of a preferred software embodiment of the arithmetic decoding, MR code frequency accumulation and MR decoding units of Fig. 1.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Reference is now made to Fig. 1 which is a simplified block diagram of an image manipulation system constructed and operative in accordance with a preferred embodiment of the present invention.

As shown, a digital representation of an image is provided from any suitable source, such as a scanner 10 which scans a substrate such as a continuous level photograph 20, a digital camera 30, a fax machine 40, an image creation workstation 50 such as a Macintosh equipped with the Adobe Photoshop software package, or a storage medium such as a hard disk 60. The digital representation of the image may be either a continuous level image or a discrete level image such as a document or other black and white image.

If the digital representation of the image is not binary, the digital representation is binarized, as indicated in Fig. 1 by image binarization block 70, using any conventional binarizing technique such as those described in Foley, J. et al, Computer Graphics: Principles and practice, 2nd Ed., Section 13.1.2, pages 568 - 573.

The binarized image is then coded by MR coding unit 80, using the MR coding scheme described in CCITT Recommendation T.4 and T.6 for Groups 3 or 4.

The MR coded binarized image generated by MR coding unit 80 then undergoes arithmetic coding in arithmetic coding unit 90. The arithmetic coding unit 90 receives as input:

a. the sequence of MR code elements which forms the MR coded binarized image and

b. the estimated probability of each MR code element, which is provided by an MR code element frequency accumulation unit 100. Initially, the estimated probabilities of all MR code elements are typically taken to be equal. However, as the MR code element sequence flows

into the MR code element frequency accumulation unit 100, the estimated probabilities change based on the number of times each MR code element is encountered.

The sequence of MR code elements typically includes code elements of 3 types:

a. MR control type code elements;

b. Black run length type code elements; and

c. White run length type code elements.

The frequency accumulation unit 100 typically receives as input each MR code element and, associated therewith, an indication of the type of that MR code element. Typically, unit 100 computes the relative code element frequency for each code element within its own code element type.

The arithmetic coding unit 90 may, if desired, be replaced by an entropy encoder or an adaptive Huffman encoder. If this is the case, then the arithmetic decoding unit 110, described below, is replaced by an entropy decoder or adaptive Huffman decoder, respectively.

One software embodiment of arithmetic coding unit 90 is described in "Arithmetic coding and statistical modeling", Dr. Dobb's Journal, Feb. 1991, pp. 16 - 29. The above reference also provides a software embodiment of arithmetic decoding unit 110.

An alternative implementation of MR code element frequency accumulation unit 100 is described below with reference to Fig. 2.

The output of the arithmetic coding unit 90 is a very compact representation of the original image which is suitable, for example, for compact storage on any suitable optical or magnetic medium and/or for rapid facsimile transmission, 105, on conventional equipment which preferably has a error correction capability, such as the V32bis modem.

The compact representation of the original image is decompressed after being transmitted or after

being retrieved from archival. To decompress the compact representation, the compressed data stream is fed to an arithmetic decoding unit 110 which replaces each arithmetically coded element with a corresponding MR code element according to the frequency of the arithmetically coded element. The frequency information is provided by an MR code element frequency accumulation unit 120 which is typically identical to unit 100. Initially, the estimated probabilities of all MR code elements are typically taken to be equal. However, as the MR code element sequence flows into the MR code element frequency accumulation unit 120, the estimated probabilities change based on the number of times each MR code element is encountered.

The output of the arithmetic decoding unit 110 is a sequence of MR code elements which is decoded by an MR decoding unit 130 using the MR decoding scheme described in CCITT Recommendation T.4 and T.6 for Groups 3 or 4.

The output of MR decoding unit 130 is a decompressed binarized image which is substantially identical to the binarized image generated by image binarization unit 70. Fig. 2 is a simplified flowchart illustrating a preferred mode of operation in which either of the MR code element frequency accumulation units 100 or 120 of Fig. 1 processes a single MR code element in a sequence of MR code elements.

If (process 210) there is a decision to reset, i.e. to begin accumulating frequencies from zero, then the method advances to stage 220. Otherwise, the method advances to stage 240. A reset is performed, for example, if a new image is to be processed whose characteristics are thought to differ significantly from the previous image processed.

In process 220, a table is allocated for each of the three MR code element types. The number of cells

in each table typically exceeds the number of code elements of each type, by 1. The difference between the content of the i'th cell in the table and the (i+1)th cell in the table, also termed herein "the i'th interval", is indicative of the relative frequency of the i'th code element, within its code element type.

Since there are 92 code elements of the White Run Length type and of the Black Run Length type, the tables for these two types each typically have 93 cells. Since there are 9 code elements of the MR Control type, the table for the MR Control type typically has 10 cells.

PROCESS 230: The table contents are initialized by generating equal intervals such as, typically, intervals having a length of 1.

PROCESS 240: Input is received: A single MR code element from the MR code element sequence representing the image, and, associated therewith, its MR code element type, is received as input.

PROCESS 250: Unit 100 allows arithmetic coder 90 to arithmetically code the current MR code element, by supplying the frequency intervals stored in the table corresponding to the current MR code element to the arithmetic coder 90. For example, if the MR code element is of the MR_control type, the intervals stored in the MR_control table are employed.

Unit 120 allows the decoder 110 to arithmetically decode the current MR code element, by supplying the same information to decoder 110.

PROCESS 260: The appropriate table is updated by incrementing by 1 the contents of each cell starting from the cell following the cell corresponding to the current code element.

For example, if the fourth MR_control type code element is encountered, the contents of the fifth to ninth cells of the MR-control table are incremented by 1.

Preferably, old frequency information is given

less weight than new frequency information. One implementation of this rule is:

PROCESS 270: For each type t, each time N_t code elements of type t have been processed, divide the cell contents of the frequency interval table of type t, by a suitable number such as 2. Suitable N_t values are: 256 for MR control type, 2048 for black and white run length types.

Appendix A is a computer listing in C language, of a preferred software embodiment of the MR coding, arithmetic coding and MR code element frequency accumulation units of Fig. 1.

Appendix B is a computer listing in C language, of a preferred software embodiment of the arithmetic decoding, MR code element frequency accumulation and MR decoding units of Fig. 1.

The programs listed in Appendices A and B may be run on a conventional computer such as any UNIX computer.

It is appreciated that the MR coding described hereinabove may, alternatively be replaced by MMR coding or other similar coding schemes.

It is appreciated that the invention shown and described herein is suitable for compressing and decompressing any type of binarized image, such as binarized discrete level images or binarized continuous level images, also termed herein "halftone images".

In certain applications, it may be desirable to use the compression methods shown and described herein to compress only a portion of a binarized image. For example, in medical imaging applications, the compression methods shown and described herein may be employed to generally losslessly compress the foreground of the medical image whereas the background of the medical image may be compressed using lossy techniques.

It is appreciated that the software components of the present invention may, if desired, be implemented in ROM (read-only memory) form. The software components may, generally, be implemented in hardware, if desired, using conventional techniques.

It is appreciated that the particular embodiment described in the Appendices is intended only to provide an extremely detailed disclosure of the present invention and is not intended to be limiting.

It is appreciated that various features of the invention which are, for clarity, described in the contexts of separate embodiments may also be provided in combination in a single embodiment. Conversely, various features of the invention which are, for brevity, described in the context of a single embodiment may also be provided separately or in any suitable subcombination.

It will be appreciated by persons skilled in the art that the present invention is not limited to what has been particularly shown and described hereinabove. Rather, the scope of the present invention is defined only by the claims that follow:

APPENDIX A

C:\ARIK\COMPRESS\PTNTSRC\AGCMP.C - Thu Aug 25 09:03:04 1994

```
/*****************************************************************
AGCMP COMPRESSION UTILITY
===========================

The following sources implement the suggested compression technique
previously described.

The agcmp program compresses a raw binary file (with no headers and with
a known line length) to a compressed file on the disk.

FILES:
======
agcmp.c - the main loop for compression. Converts the raw file to
      MR codes and passes them to the arithmetic coder.


The following sources are common to both programs - agcmp and
agexp (Decompression) and handle the statistical estimation
(element frequency accumulation) and the arithmetic coding:

amdl.c - Statistical estimation. Based on a source from Dr. Dobbs
      journal, February 1991, "Arithmatic Coding and Statistical
      Modeling" by Mark R. Nelson, but modified to fit compression
      of MR codes.

acoder.c, abitio.c - Implement the arithmetic coder, based on Dr. Dobbs
            Journal.

COMPILATION:
============
agcmp: cc agcmp.c amdl.c acoder.c abitio.c

FURTHER INFORMATION about agcmp.c:
==================================

AUTHOR: Arik Gordon

INPUT: A rastered file (No headers!) with 1728 binary pixels per line
OUPUT: compressed file.
USAGE: agcmp IN_FILE OUT_FILE
Desc : This source opens a rastered binary file, converts it to codes
     according to MR standard, and passes the codes to the arithmatic
     coder. The compressed file is constructed from a header (see agcmp.h)
     and the compressed entropy coded stream.
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <dos.h>
#include "acoder.h"
#include "amodel.h"
```

```
#include "abitio.h"
#include "agcmp.h"

static char *last_line_in_prev_strip;

long agcmp(char *infile, char *outfile);       // returns size in bytes
long add_file(char *in, int out);
long mr_compress_strip(char bufi[], int lines);
void   modified_READ(char *prev,char *cur, char *next, int length);
void   one_line_modified_read(char *prev, char  *curr, int length);
void   put_rl(int len, int color);
void   put_code(int len, int color);
void put_EOL();
  find_next(int color, int pos, char  *line, int len);
void   erase_single_dots(char *prev, char *curr, char *next, int len);

main(int argc, char *argv[])
{
   if (argc != 3) {
      fprintf(stderr, "\nUsage: %s IMG_file_name G3_output_file_name\n", argv[0]);
      exit(9);
   }
   printf("total_bytes = %ld\n", agcmp(argv[1], argv[2]));
}


long agcmp(char *infile, char *outfile)      // returns size in bytes
{
   long total_bytes=0L;
   unsigned int i, j=0, k, file_count;
   char *bufi;
   unsigned size_in_bytes;
   AG_HEADER ag_header;
   int fdi, fdtmp;

   if ((fdi=open(infile, O_RDONLY | O_BINARY, S_IREAD | S_IWRITE)) < 1)
      BigErr(9, "cmr: Can't Open");

   /* INITIALIZE ARITHMETIC CODER */
   initialize_model();
   init_mr_model();
   initialize_output_bitstream(outfile, &ag_header, sizeof(AG_HEADER));
   initialize_arithmetic_encoder();

   if ( (bufi = malloc(STRIP_SIZE*BYTES_PER_LINE)) == NULL )
      BigErr(9, "AGCMP: no mem");

   if ( (last_line_in_prev_strip = malloc(PELS_PER_LINE)) == NULL)
      BigErr(9, "agcmp1: no mem");

   memset(last_line_in_prev_strip, 0, PELS_PER_LINE);
   ag_header.number_of_lines_in_file = 0;

   /* Main loop for Compression */
   while ((file_count=read(fdi, bufi, STRIP_SIZE*BYTES_PER_LINE)) >= BYTES_PER_LINE) {
      fprintf(stderr, "COMPRESSING STRIP #%d\r", j++);
```

C:\ARIK\COMPRESS\PTNTSRC\AGCMP.C - Thu Aug 25 09:03:04 1994

```
    ag_header.number_of_lines_in_file += file_count/BYTES_PER_LINE;
    mr_compress_strip(bufi, file_count/BYTES_PER_LINE);
    _heapmin();
  }
  fprintf(stderr, "\n");
  free(last_line_in_prev_strip);
  free(bufi);
  close(fdi);
  _heapmin();

  /* Finish and close arithmetic coding */
  code_EOF();
  flush_arithmetic_encoder();
  total_bytes = flush_output_bitstream(&ag_header, sizeof(AG_HEADER));
  free_amdl_bufs();

  return(total_bytes);
}

/* compress one strip (arbitrary size, defined in agcmp.h) */

long mr_compress_strip(char bufi[], int lines)
{
   char array[3][PELS_PER_LINE];
   unsigned k, i, cur_line=2, off;

   // Fill first 2 lines in array.

   for (k=0; k < min(2, lines); k++)
      for (i=0; i<PELS_PER_LINE; i++)
         array[k+1][i] = ((bufi[k*BYTES_PER_LINE+i/8] & (1 << (7-(i%8)))) != 0);

   if (lines > 0)  // There is at least 1 line to compress
      modified_READ(NULL, &array[1][0], &array[2][0], PELS_PER_LINE);  // First array compression

   /* convert packed bits to "1 bit per byte" format */
   while (cur_line < lines) {
      memcpy(&array[0][0], &array[1][0], 2 * PELS_PER_LINE);
      for (i=0; i<PELS_PER_LINE; i++) {
         off = cur_line * BYTES_PER_LINE+i/8;
         if (bufi[off] == 0) {
            memset(&(array[2][i]), 0, 8);
            i+=7;
            continue;
         }
         if (bufi[off] == 255) {
            memset(&(array[2][i]), 1, 8);
            i+=7;
            continue;
         }
         array[2][i] = ((bufi[off] & (1 << (7-(i%8)))) != 0);
      }
      cur_line++;
      /* compress one line (given the previous line)*/
      /* (we also provide the next line in case some filtering is
         desired) */
      modified_READ(&array[0][0], &array[1][0], &array[2][0], PELS_PER_LINE);
```

C:\ARIK\COMPRESS\PTNTSRC\AGCMP.C - Thu Aug 25 09:03:04 1994

```
  /* do last line */
  if (lines > 1) {
    memcpy(&array[0][0], &array[1][0], 2 * PELS_PER_LINE);
    modified_READ(&array[0][0], &array[1][0], NULL, PELS_PER_LINE);
  }
  return(1);
}

void  modified_READ(char *prev,char *cur, char *next, int length)
{
  int j;
  long i;

  cur[0] = WHITE; // don't accept a black pixel on line beginning

  if (prev = =NULL) {
    one_line_modified_read(last_line_in_prev_strip, cur, length);
    return;
  }
  memcpy(last_line_in_prev_strip, cur, PELS_PER_LINE);
  one_line_modified_read(prev, cur, length);
}

/* Here we actuallt translate the line to MR codes + Run-Lengths
  and pass the codes to the arithmetic coder */
void  one_line_modified_read(char *prev, char  *curr, int length)
{
  int a0, a1, a2, b1, b2, a0_color;

  a0 = -1; a0_color = WHITE;

  // *curr = WHITE; // don't accept a black pixel on line beginning

  do {
    a1 = find_next(!a0_color, a0+1, curr, length);
    a2 = find_next(a0_color, a1+1, curr, length);

    if (a0 = = -1)
      b1 = find_next(!a0_color, a0+1, prev, length);
    else if (prev[a0] = = a0_color)
      b1 = find_next(!a0_color, a0+1, prev, length);
    else {
      b1 = find_next(a0_color, a0+1, prev, length);
      b1 = find_next(!a0_color, b1+1, prev, length);
    }

    b2 = find_next(a0_color, b1+1, prev, length);

    // code it
    if (b2 < a1) {   // PASS mode
      //printf("PASS (a0=%d, a1=%d, a2=%d, b1=%d, b2=%d)\n", a0, a1, a2, b1, b2);
      code_1(MR_CONTROL, PASS);
      a0 = b2;
```

C:\ARIK\COMPRESS\PTNTSRC\ACCMP.C - Thu Aug 25 09:03:04 1994

```
} else if (abs(a1-b1) <= 3) {  // VERTICAL mode
  switch (a1-b1) {
    case 0:
      //printf("V0 (a0=%d, a1=%d, a2=%d, b1=%d, b2=%d)\n", a0, a1, a2, b1, b2);
      code_1(MR_CONTROL, V0);
      break;
    case 1:
      //printf("VR1 (a0=%d, a1=%d, a2=%d, b1=%d, b2=%d)\n", a0, a1, a2, b1, b2);
      code_1(MR_CONTROL, VR1);
      break;
    case -1:
      //printf("VL1 (a0=%d, a1=%d, a2=%d, b1=%d, b2=%d)\n", a0, a1, a2, b1, b2);
      code_1(MR_CONTROL, VL1);
      break;
    case 2:
      //printf("VR2 (a0=%d, a1=%d, a2=%d, b1=%d, b2=%d)\n", a0, a1, a2, b1, b2);
      code_1(MR_CONTROL, VR2);
      break;
    case -2:
      //printf("VL2 (a0=%d, a1=%d, a2=%d, b1=%d, b2=%d)\n", a0, a1, a2, b1, b2);
      code_1(MR_CONTROL, VL2);
      break;
    case 3:
      //printf("VR3 (a0=%d, a1=%d, a2=%d, b1=%d, b2=%d)\n", a0, a1, a2, b1, b2);
      code_1(MR_CONTROL, VR3);
      break;
    case -3:
      //printf("VL3 (a0=%d, a1=%d, a2=%d, b1=%d, b2=%d)\n", a0, a1, a2, b1, b2);
      code_1(MR_CONTROL, VL3);
      break;

  }
  a0 = a1;
 } else {  // HORIZONTAL MODE
  if (a0 == -1)
    a0 = 0;
  //printf("HORIZONTAL: COLOR = %d, LEN1 = %d, LEN2 = %d (a0=%d)\n", a0_color, a1-a0, a2
===> -a1, a0);
  code_1(MR_CONTROL, HOR);
  put_rl(a1-a0, a0_color);
  put_rl(a2-a1, !a0_color);
  a0 = a2;
 }
 if (a0 < length)
   a0_color = curr[a0];
} while (a0 < length);
//printf("EOL\n");
//put_EOL(line);  /* we don't need it because next a0 is beyond line */
}

/* converts a single run-length (unlimited length) to several runs
  according to MR (Group3,4) spec */
void  put_rl(int len, int color)
{
  if (len > 63) {
    put_code((len / 64) + 63, color);
    len -= (len / 64) * 64;
```

C:\ARIK\COMPRESS\PTNTSRC\AGCMP.C - Thu Aug 25 09:03:04 1994

```
  }
  put_code(len, color);
}

/* codes one legitimate run */
void  put_code(int len, int color)
{
  code_1(color, BW_SYMBOLS - len - 1);
}

/* We do not need this if we know the line length in advance */
void put_EOL()
{
  //code_1(WHITE, EOL);
  //code_1(BLACK, EOL);
}

/* finds the next color interchange */
find_next(int color, int pos, char  *line, int len)
{
  int i;
  char *ptr;

  if (pos > len-1)
     return(len);

  if ( (ptr = memchr(line+pos, color, len-pos)) == NULL)
     return len;
  else
     return (ptr-line);
}

BigErr(int n, char *s)  // too many bits in strip.
{
  printf("Err %d -- %s", n, s);
  exit(9);
}

/* codes 1 symbol (Control or Black Run or White Run) */
code_1(int mode, int c)
{
  SYMBOL s;

  convert_int_to_symbol( c, &s, mode);
  encode_symbol( &s );
  update_model( c );
}

/* to finish with the arithmetic coding: */
code_EOF()
{
  SYMBOL s;

  convert_int_to_symbol( EOF, &s, MR_CONTROL);
  encode_symbol( &s );
}
```

C:\ARIK\COMPRESS\PTNTSRC\AGCMP.H - Thu Aug 25 09:03:46 1994

```
/* Desc: Header file mainly for agcmp.c, agexp.c */
/* AUTHOR: Arik Gordon                    */


/* This is a header that appears at the begining of  the compressed file */
typedef struct AG_HEADER {
   long total_bytes;
   long number_of_lines_in_file;
} AG_HEADER;


/* In our implementaion we assume a standard fax document with 1728 pixels
   per line */
#define  PELS_PER_LINE      1728
#define  BYTES_PER_LINE      216


#define STRIP_SIZE 100

#define WHITE 0
#define BLACK 1
#define MR_CONTROL 2

#define MR_SYMBOLS 9
#define BW_SYMBOLS 93

#define V0    8
#define PASS  2
#define VL1   3
#define VR1   4
#define HOR   5
#define VL2   6
#define VL3   7
#define VR2   1
#define VR3   0


#define beep0     putch(7)
```

C:\ARIK\COMPRESS\PTNTSRC\AMDL.C - Thu Aug 25 09:04:58 1994

```
/*
 * Listing 9 -.amdl.c
 *
 * AUTHOR: Originally from Dr. Dobbs, Feb 1991, Substantially modified
 *       by Arik Gordon.
 *
 * This is the statistical estimation module for compressing
 * MR codes. There are three types ot codes: MR_CONTROL, BLACK Run-Length
 * and WHITE Run-Length. For each type we have a seperate statistical
 * estimator of order 0 for run-lengths and order 2 for MR_CONTROL
 *
 * This is a relatively simple model. For each symbol type,
 * the totals for all of the symbols are stored in an corresponding
 * array (e.g. *mr_storage*). This array has valid indices from -1
 * to Ni. The reason for having a -1 element is because the EOF
 * symbols is included in the table, and it has a value of -1.
 * (Ni = number of different symbol for each type)
 *
 * The total count for all the symbols is stored in totals[Ni], and
 * the low and high counts for symbol c are found in *array*[c] and
 * array[c+1].
 */

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <io.h>
#include <errno.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>

#include "AGCMP.H"
#include "acoder.h"
#include "amodel.h"

/*
 * In order to create an array with indices -1 through num_of_symbols, I have
 * to do this funny declaration. totals[-1] == storage[0].
 */
short int **mr_storage;
short int *wt_storage;
short int *bl_storage;
short int *totals;
static int num_of_symbols, maximum_scale;
static int prev, prev1;

/*
 * When the model is first started up, each symbols has a count of
 * 1, which means a low value of c+1, and a high value of c+2.
 */
void initialize_model()
{
   int i, j, order_2_symbols;

   prev = prev1 = 0;
   num_of_symbols = MR_SYMBOLS;
```

C:\ARIK\COMPRESS\PTNTSRC\AMDL.C - Thu Aug 25 09:04:58 1994

```
order_2_symbols = num_of_symbols * num_of_symbols;
mr_storage = (int **) malloc(sizeof(int *) * (order_2_symbols+1));
for (i=0; i<order_2_symbols; i++)
  mr_storage[i] = malloc(sizeof(int) * (num_of_symbols+2));

for (j=0; j<order_2_symbols; j++) {
 totals = &(mr_storage[j][1]);
 for ( i = -1 ; i <= num_of_symbols ; i++ )
  totals[ i ] = i + 1;
}

num_of_symbols = BW_SYMBOLS;
wt_storage = malloc((num_of_symbols+2) * sizeof(int));
totals = &(wt_storage[1]);

for ( i = -1 ; i <= num_of_symbols ; i++ )
  totals[i] = i + 1;


bl_storage = malloc((num_of_symbols+2) * sizeof(int));
totals = &(bl_storage[1]);

for ( i = -1 ; i <= num_of_symbols ; i++ )
  totals[i] = i + 1;
}

/*
 * Updating the model means incrementing every single count from
 * the high value for the symbol on up to the total. Then, there
 * is a complication. If the cumulative total has gone up to
 * the maximum value, we need to rescale. Fortunately, the rescale
 * operation is relatively rare.
 */
void  update_model( int symbol )
{
  int i;

  for ( symbol++ ; symbol <= num_of_symbols; symbol++ )
    totals[ symbol ]++;
  if ( totals[ num_of_symbols ] == maximum_scale )
  {
    for ( i = 0 ; i <= num_of_symbols ; i++ )
    {
      totals[ i ] /= 2;
      if ( totals[ i ] <= totals[ i-1 ] )
        totals[ i ] = totals[ i-1 ] + 1;
    }
  }
}

/*
 * Finding the low count, high count, and scale for a symbol
 * is really easy, because of the way the totals are stored.
 * This is the one redeeming feature of the data structure used
 * in this implementation.
 */
```

C:\ARIK\COMPRESS\PTNTSRC\AMDL.C - Thu Aug 25 09:04:58 1994

```
int   convert_int_to_symbol( int c, SYMBOL *s, int mode )
{
  switch(mode) {
    case WHITE:
      totals = wt_storage + 1;
      num_of_symbols = BW_SYMBOLS;
      maximum_scale = 2048;
      break;
    case BLACK:
      totals = bl_storage + 1;
      num_of_symbols = BW_SYMBOLS;
      maximum_scale = 2048;
      break;
    case MR_CONTROL:
      num_of_symbols = MR_SYMBOLS;
      totals = mr_storage[(prev1 * num_of_symbols + prev)] + 1;
      prev1 = prev;
      prev = c;
      maximum_scale = 256;
      break;
  }

  s->scale = totals[ num_of_symbols ];
  s->low_count = totals[ c ];
  s->high_count = totals[ c+1 ];
  return( 0 );
}

/*
 * Getting the scale for the current context is easy.
 */
void   get_symbol_scale( SYMBOL *s, int mode, int prev, int prev1)
{
  switch(mode) {
    case WHITE:
      totals = wt_storage + 1;
      num_of_symbols = BW_SYMBOLS;
      maximum_scale = 2048;
      break;
    case BLACK:
      totals = bl_storage + 1;
      num_of_symbols = BW_SYMBOLS;
      maximum_scale = 2048;
      break;
    case MR_CONTROL:
      num_of_symbols = MR_SYMBOLS;
      totals = mr_storage[(prev1 * num_of_symbols + prev)] + 1;
      maximum_scale = 256;
      break;
  }
  s->scale = totals[ num_of_symbols ];
}

/*
 * During decompression, we have to search through the table until
 * we find the symbol that straddles the "count" parameter.  When
 * it is found, it is returned. The reason for also setting the
```

C:\ARIK\COMPRESS\PTNTSRC\AMDL.C - Thu Aug 25 09:04:58 1994

```
* high count and low count is so that symbol can be properly removed
* from the arithmetic coded input.
*/
int  convert_symbol_to_int( int count, SYMBOL *s)
{
  int c;

  for ( c = num_of_symbols-1; count < totals[ c ] ; c-- )
    ;
  s->high_count = totals[ c+1 ];
  s->low_count = totals[ c ];
  return( c );
}

/* The following is an optional module, that initializes the statistical
  estimation tables with pre-defined values. It can slightly improve
  compression of small files */

init_mr_model()
{
  int i;

  update_initial_mr_model( V0, 6);
  update_initial_mr_model( VL1, 2);
  update_initial_mr_model( VR1, 2);
  update_initial_mr_model( HOR, 2);
  update_initial_mr_model( PASS, 1);
}

update_initial_mr_model( int symbol, int count )
{
  int i, prev, prev1, j;

  num_of_symbols = MR_SYMBOLS;
  maximum_scale = 256;

  for (prev = 0; prev<num_of_symbols; prev++)
   for (prev1 = 0; prev1<num_of_symbols; prev1++) {
    totals = mr_storage[(prev1 * num_of_symbols + prev)] + 1;
    for (j=0; j<count; j++)
     update_model(symbol);
   }
}

free_amdl_bufs()
{
  int i, order_2_symbols;

  num_of_symbols = MR_SYMBOLS;
  order_2_symbols = num_of_symbols * num_of_symbols;
  for (i=0; i<order_2_symbols; i++)
    free(mr_storage[i]);
  free(mr_storage);

  num_of_symbols = BW_SYMBOLS;
  free(wt_storage);
```

C:\ARIK\COMPRESS\PTNTSRC\AMD.C - Thu Aug 25 09:04:58 1994

```
  free(bl_storage);
  // _heapmin();
}
```

C:\ARIK\COMPRESS\PTNTSRC\AMODEL.H - Thu Aug 25 08:58:06 1994

```
/*
 * Listing 8 -- amodel.h
 *
 * This file contains all of the function prototypes and
 * external variable declarations needed to interface with
 * the modeling code found in amdl.c.
 */

/*
 * Eternal variable declarations.
 */
extern int max_order;
extern int flushing_enabled;
/*
 * Prototypes for routines that can be called from MODEL-X.C
 */
void initialize_model( void );
void update_model( int symbol );
int  convert_int_to_symbol( int symbol, SYMBOL *s, int mode );
void get_symbol_scale( SYMBOL *s, int mode, int prev, int prev1 );
int  convert_symbol_to_int( int count, SYMBOL *s );
void add_character_to_model( int c );
void flush_model( void );
```

C:\ARIK\COMPRESS\PTNTSRC\ACODER.C - Thu Aug 25 09:11:38 1994

```
/*
 * Listing 2 - coder.c
 *
 * SOURCE: Dr. Dobbs Journal, Feb 1991 + minor modifications by
 *      Arik Gordon
 *
 * This file contains the code needed to accomplish arithmetic
 * coding of a symbol. All the routines in this module need
 * to know in order to accomplish coding is what the probabilities
 * and scales of the symbol counts are. This information is
 * generally passed in a SYMBOL structure.
 *
 * This code was first published by Ian H. Witten, Radford M. Neal,
 * and John G. Cleary in "Communications of the ACM" in June 1987,
 * and has been modified slightly.
 */

#include <stdio.h>
#include "acoder.h"
#include "abitio.h"
#include "AGCMP.H"

/*
 * These four variables define the current state of the arithmetic
 * coder/decoder. They are assumed to be 16 bits long. Note that
 * by declaring them as short ints, they will actually be 16 bits
 * on most 80X86 and 680X0 machines, as well as VAXen.
 */


static unsigned short int code; /* The present input code value     */
static unsigned short int low;  /* Start of the current code range  */
static unsigned short int high; /* End of the current code range    */
long underflow_bits;            /* Number of underflow bits pending */

/*
 * This routine must be called to initialize the encoding process.
 * The high register is initialized to all 1s, and it is assumed that
 * it has an infinite string of 1s to be shifted into the lower bit
 * positions when needed.
 */
void initialize_arithmetic_encoder()
{
   low = 0;
   high = 0xffff;
   underflow_bits = 0;
}

/*
 * This routine is called to encode a symbol. The symbol is passed
 * in the SYMBOL structure as a low count, a high count, and a range,
 * instead of the more conventional probability ranges. The encoding
 * process takes two steps. First, the values of high and low are
 * updated to take into account the range restriction created by the
 * new symbol. Then, as many bits as possible are shifted out to
 * the output stream. Finally, high and low are stable again and
 * the routine returns.
```

C:\ARIK\COMPRESS\PTNTSRC\ACODER.C - Thu Aug 25 09:11:38 1994

```
*/

void _fastcall encode_symbol( SYMBOL *s )
{
  long range;
/*
 * These three lines rescale high and low for the new symbol.
 */
   range = (long) ( high-low ) + 1;
   high = low + (unsigned short int )
          (( range * s->high_count ) / s->scale - 1 );
   low = low + (unsigned short int )
          (( range * s->low_count ) / s->scale );
/*
 * This loop turns out new bits until high and low are far enough
 * apart to have stabilized.
 */
   for ( ; ; )
   {
/*
 * If this test passes, it means that the MSDigits match, and can
 * be sent to the output stream.
 */
      if ( ( high & 0x8000 ) == ( low & 0x8000 ) )
      {
      output_bit(high & 0x8000 );
         while ( underflow_bits > 0 )
         {
      output_bit( ~high & 0x8000 );
            underflow_bits--;
         }
      }
/*
 * If this test passes, the numbers are in danger of underflow, because
 * the MSDigits don't match, and the 2nd digits are just one apart.
 */
      else if ( ( low & 0x4000 ) && !( high & 0x4000 ))
      {
         underflow_bits += 1;
         low &= 0x3fff;
         high |= 0x4000;
      }
      else
      return;
      low <<= 1;
      high <<= 1;
      high |= 1;
   }
}

/*
 * At the end of the encoding process, there are still significant
 * bits left in the high and low registers.  We output two bits,
 * plus as many underflow bits as are necessary.
 */
void flush_arithmetic_encoder()
```

C:\ARIK\COMPRESS\PTNTSRC\ACODER.C - Thu Aug 25 09:11:38 1994

```
{
  output_bit(low & 0x4000 );
  underflow_bits++;
  while ( underflow_bits-- > 0 )
  output_bit(~low & 0x4000 );
}

/*
 * When decoding, this routine is called to figure out which symbol
 * is presently waiting to be decoded. This routine expects to get
 * the current model scale in the s->scale parameter, and it returns
 * a count that corresponds to the present floating point code:
 *
 *  code = count / s->scale
 */
int  get_current_count( SYMBOL *s )
{
  long range;
  short int count;


  range = (long) ( high - low ) + 1;
  count = (short int)
       ((((long) ( code - low ) + 1 ) * s->scale-1 ) / range );
  return( count );
}

/*
 * This routine is called to initialize the state of the arithmetic
 * decoder. This involves initializing the high and low registers
 * to their conventional starting values, plus reading the first
 * 16 bits from the input stream into the code value.
 */
void initialize_arithmetic_decoder()
{
  int i;

  code = 0;
  for ( i = 0 ; i < 16 ; i++ )
  {
    code <<= 1;
  code += input_bit();
  }
  low = 0;
  high = 0xffff;
}

/*
 * Just figuring out what the present symbol is doesn't remove
 * it from the input bit stream. After the character has been
 * decoded, this routine has to be called to remove it from the
 * input stream.
 */
void remove_symbol_from_stream(SYMBOL *s )
{
  long range;
```

C:\ARIK\COMPRESS\PTNTSRC\ACODER.C - Thu Aug 25 09:11:38 1994

```
/*
 * First, the range is expanded to account for the symbol removal.
 */

  range = (long)( high - low ) + 1;
  high = low + (unsigned short int)
          (( range * s->high_count ) / s->scale - 1 );
  low = low + (unsigned short int)
          (( range * s->low_count ) / s->scale );
/*
 * Next, any possible bits are shipped out.
 */
  for ( ; ; )
  {
/*
 * If the MSDigits match, the bits will be shifted out.
 */
     if ( ( high & 0x8000 ) == ( low & 0x8000 ) )
     {
     }
/*
 * Else, if underflow is threatining, shift out the 2nd MSDigit.
 */
     else if ((low & 0x4000) == 0x4000  && (high & 0x4000) == 0 )
     {
        code ^= 0x4000;
        low  &= 0x3fff;
        high |= 0x4000;
     }
/*
 * Otherwise, nothing can be shifted out, so I return.
 */
      else
      return;
      low <<= 1;
      high <<= 1;
      high |= 1;
      code <<= 1;
    code += input_bit0;
    }
  }
```

C:\ARIK\COMPRESS\PTNTSRC\ACODER.H - Thu Aug 25 08:57:10 1994

```
/*
 * Listing 1 -- acoder.h
 *
 * This header file contains the constants, declarations, and
 * prototypes needed to use the arithmetic coding routines.  These
 * declarations are for routines that need to interface with the
 * arithmetic coding stuff in acoder.c
 *
 */

#define MAXIMUM_SCALE   2048 // 16383  /* Maximum allowed frequency count */
#define ESCAPE          256   /* The escape symbol           */
#define DONE            -1    /* The output stream empty  symbol */
#define FLUSH           -2    /* The symbol to flush the model   */

/*
 * A symbol can either be represented as an int, or as a pair of
 * counts on a scale.  This structure gives a standard way of
 * defining it as a pair of counts.
 */
typedef struct {
          unsigned short int low_count;
          unsigned short int high_count;
          unsigned short int scale;
         } SYMBOL;

extern long underflow_bits;   /* The present underflow count in  */
                 /* the arithmetic coder.        */
/*
 * Function prototypes.
 */
void initialize_arithmetic_decoder();
void remove_symbol_from_stream( SYMBOL *s );
void initialize_arithmetic_encoder( void );
void encode_symbol( SYMBOL *s );
void flush_arithmetic_encoder();
int  get_current_count( SYMBOL *s);
```

C:\ARIK\COMPRESS\PTNTSRC\ABITIO.C - Thu Aug 25 09:12:46 1994

```
/*
 * Listing 4 -- abitio.c
 *
 *
 * SOURCE: Dr. Dobbs Journal, Feb 1991 + minor modifications by
 *       Arik Gordon
 * This routine contains a set of bit oriented i/o routines
 * used for arithmetic data compression. The important fact to
 * know about these is that the first bit is stored in the msb of
 * the first byte of the output, like you might expect.
 *
 * Both input and output maintain a local buffer so that they only
 * have to do block reads and writes. This is done in spite of the
 * fact that C standard I/O does the same thing. If these
 * routines are ever ported to assembly language the buffering
 * will come in handy.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include "acoder.h"
#include "abitio.h"

#include "ACCMP.H"

#define BUFFER_SIZE 8192
static char *buffer;              /* This is the i/o buffer    */
static char *current_byte;        /* Pointer to current byte   */

static int output_mask;           /* During output, this byte  */
                     /* contains the mask that is */
                     /* applied to the output byte*/
                     /* if the output bit is a 1  */

static int input_bytes_left;      /* During input, these three */
static int input_bits_left;       /* variables keep track of my*/
static int past_eof;              /* input state. The past_eof*/
                     /* byte comes about because */
                     /* of the fact that there is */
static long total_bytes;          /* a possibility the decoder */
                     /* can legitimately ask for  */
                     /* more bits even after the  */
                     /* entire file has been      */
                     /* sucked dry.               */

static FILE *stream;


/*
 * This routine is called once to initialze the output bitstream.
 * All it has to do is set up the current_byte pointer, clear out
 * all the bits in my current output byte, and set the output mask
 * so it will set the proper bit next time a bit is output.
 */
void initialize_output_bitstream(char *file, void *header, unsigned int header_size)
{
  buffer = malloc(BUFFER_SIZE+2);
```

C:\ARIK\COMPRESS\PTNTSRC\ABITIO.C - Thu Aug 25 09:12:46 1994

```
  if (buffer == NULL) {
     printf("\niobs:no mem\n");
     exit(9);
  }
  total_bytes = 0L;
  current_byte = buffer;
  *current_byte = 0;
  output_mask = 0x80;
  stream = fopen(file, "wb");
  setvbuf( stream, NULL, _IOFBF, 8192 );
  total_bytes += fwrite(header, 1, header_size, stream);
  //printf("total_bytes = %ld\n", total_bytes);
}

/*
 * The output bit routine just has to set a bit in the current byte
 * if requested to. After that, it updates the mask. If the mask
 * shows that the current byte is filled up, it is time to go to the
 * next character in the buffer. If the next character is past the
 * end of the buffer, it is time to flush the buffer.
 */

void  output_bit(int bit)
{
   if ( bit )
      *current_byte |= output_mask;
   output_mask >>= 1;
   if ( output_mask == 0 )
   {
      output_mask = 0x80;
      current_byte++;
      if ( current_byte == ( buffer + BUFFER_SIZE ) )
      {
         total_bytes += fwrite( buffer, 1, BUFFER_SIZE, stream );
         current_byte = buffer;
      }
      *current_byte = 0;
   }
}

/*
 * When the encoding is done, there will still be a lot of bits and
 * bytes sitting in the buffer waiting to be sent out. This routine
 * is called to clean things up at that point.
 */
long flush_output_bitstream(void *header, unsigned int header_size)
{
   total_bytes += fwrite( buffer, 1, (size_t)( current_byte - buffer ) + 1, stream );
   current_byte = buffer;
   fseek(stream, 0L, SEEK_SET);
   memcpy (header, &total_bytes, sizeof(long));
   fwrite(header, header_size, 1, stream);
   fclose(stream);
   free(buffer);
   _heapmin();
   return(total_bytes);
```

```
}

/*
 * Bit oriented input is set up so that the next time the input_bit
 * routine is called, it will trigger the read of a new block. That
 * is why input_bits_left is set to 0.
 */
void initialize_input_bitstream(char *file, void *header, unsigned int header_size)
{
  buffer = malloc(BUFFER_SIZE+2);
  if (buffer == NULL) {
     printf("\niibs:no mem\n");
     exit(9);
  }
  input_bits_left = 0;
  input_bytes_left = 1;
  past_eof = 0;
  stream = fopen(file, "rb");
  setvbuf( stream, NULL, _IOFBF, 8192 );
  fread(header, 1, header_size, stream);
}

close_input_bitstream()
{
   free(buffer);
   _heapmin();
   fclose(stream);
}

/*
 * This routine reads bits in from a file. The bits are all sitting
 * in a buffer, and this code pulls them out, one at a time. When the
 * buffer has been emptied, that triggers a new file read, and the
 * pointers are reset. This routine is set up to allow for two dummy
 * bytes to be read in after the end of file is reached. This is because
 * we have to keep feeding bits into the pipeline to be decoded so that
 * the old stuff that is 16 bits upstream can be pushed out.
 */
int input_bit()
{
  if ( input_bits_left == 0 )
  {
    current_byte++;
    input_bytes_left--;
    input_bits_left = 8;
    if ( input_bytes_left == 0 )
    {
      input_bytes_left = fread( buffer, 1, BUFFER_SIZE, stream );
      if ( input_bytes_left == 0 )
      {
        if ( past_eof )
        {
          fprintf( stderr, "Bad input file\n" );
          exit( -1 );
        }
        else
        {
```

C:\ARIK\COMPRESS\PTNTSRC\ABITIO.C - Thu Aug 25 09:12:46 1994

```
          past_eof = 1;
          input_bytes_left = 2;
        }
      }
      current_byte = buffer;
    }
  }
  input_bits_left--;
  return ( ( *current_byte >> input_bits_left ) & 1 );
}
```

C:\ARIK\COMPRESS\PTNTSRC\ABITIO.H - Thu Aug 25 09:12:56 1994

```
/*
 * Listing 3 - abitio.h
 *
 * This header file contains the function prototypes needed to use
 * the bitstream i/o routines.
 *
 */

int input_bit();
void initialize_output_bitstream(char *file, void *header, unsigned int header_size);
long flush_output_bitstream(void *header, unsigned int header_size);
void output_bit(int bit);
void initialize_input_bitstream(char *file, void *header, unsigned int header_size);
```

APPENDIX B

C:\ARIK\COMPRESS\PTNTSRC\AGEXP.C - Sun Aug 28 07:04:42 1994

```
/*********************************************************************
AGEXP DECOMPRESSION UTILITY
========================

The agexp program decompresses files created by agcmp to
a binary ratsterized file (no headers) on the disk.

(File size is fixed and determined in agcmp.h)

FILES:
======
 agexp.c - the main loop for decompression. Retrieves MR codes
       from the arithmetic coder and re-generates the raw
       binary file.


 The following sources are common to both programs - agcmp and
 agexp (Decompression) and handle the statistical estimation
 (element frequency accumulation) and the arithmetic coding:

 amdl.c - Statistical estimation. Based on a source from Dr. Dobbs
       journal, February 1991, "Arithmatic Coding and Statistical
       Modeling" by Mark R. Nelson, but modified to fit compression
       of MR codes.

 acoder.c, abitio.c - implement the arithmetic coder, based on Dr. Dobbs
             Journal.

 COMPILATION:
 ============
 agexp: cc agexp.c amdl.c acoder.c abitio.c

 FURTHER INFORMATION about agexp.c:
 ==================================

 AUTHOR: Arik Gordon

 INPUT: compressed file.
 OUPUT: A rastered file (No headers!) with 1728 binary pixels per line
 USAGE: agexp COMPRESSED_FILE_NAME   RASTER_FILE_NAME
 Desc : This is the main loop for agexp utility. It makes calls to the
     arithmetic coder to retrieve the MR codes, and than builds a
     ratered binary image.
 *********************************************************************/


 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
 #include <fcntl.h>
 #include <memory.h>
 #include <malloc.h>
 #include <sys\types.h>
 #include <sys\stat.h>
 // #include <dos.h>
```

C:\ARIK\COMPRESS\PTNTSRC\AGEXP.C - Sun Aug 28 07:04:42 1994

```
#include "acoder.h"
#include "amodel.h"
#include "abitio.h"
#include "agcmp.h"

 find_next(int color, int pos, char *line, int len);
int uncompress_strip_and_save(unsigned char *compressed,long compressed_size,int fdo);
 mr_uncompress(unsigned char *line, unsigned char *prev);
void huf_uncompress(unsigned char *line,unsigned char *str);
void get_bit_stream(char *str,char *bufi,long compressed_size);
int pack8(unsigned char *line,unsigned char *buf);
int find_b1(int a0_color,int a0,char *line,int length);
void vertical_code(int *a0_color,int *a0,char *prev,int length,char *curr,int offset);
int find_huf_len(int *a0_color);


#define STRIP_SIZE 100 // can be any number, determines buffer size


main(int argc, char *argv[])
{
  if (argc != 3) {
   fprintf(stderr, "\nUsage: %s G3_output_file_name IMG_file_name \n", argv[0]);
   exit(9);
  }
  agexp(argv[1], argv[2]);
}

agexp(char *infile, char *outfile)
{
  char line[PELS_PER_LINE], prev_line[PELS_PER_LINE], *bufo;
  int fdo;
  unsigned char *compressed;
  long compressed_size;  // in bits
  int j=0, line_num = 0;
  AG_HEADER ag_header;


  if ((fdo=open(outfile, O_WRONLY | O_CREAT | O_TRUNC | O_BINARY, S_IREAD | S_IWRITE)) < 1)
  BigErr(9, "AGEXP: Can't open outfile");

  if ( (bufo = malloc(STRIP_SIZE*BYTES_PER_LINE)) == NULL )
  BigErr(9, "AGEXP: no mem");

  memset(prev_line, 0, PELS_PER_LINE);

  initialize_model();
  init_mr_model();
  initialize_input_bitstream(infile, &ag_header, sizeof(ag_header));
  initialize_arithmetic_decoder();
  init_get_10;

  printf("LINES: %ld   TOTAL: %ld\n", ag_header.number_of_lines_in_file, ag_header.total_bytes);

  while ( mr_uncompress(line, prev_line) != -1 ) {
  if ( line_num++ % 100 == 0)
    printf("line %d\r", line_num-1);
```

C:\ARIK\COMPRESS\PTNTSRC\AGEXP.C - Sun Aug 28 07:04:42 1994

```
  memcpy(prev_line, line, PELS_PER_LINE);
  pack8(line, bufo+j*BYTES_PER_LINE);
  j++;
  if (j == STRIP_SIZE) {
    write(fdo, bufo, BYTES_PER_LINE*STRIP_SIZE);
    j = 0;
  }
  }

  if (j != 0)
   write(fdo, bufo, BYTES_PER_LINE*j);


  free_amdl_bufs();
  close_get_1();
  free(bufo);
  close(fdo);
}


///////////////////////////////////////////////////////////////////////
/** This loop decompresses one rasterized line ! **/
mr_uncompress(unsigned char *line, unsigned char *prev)
{
  int a0_color = WHITE, b1, b2;
  int a0 = 0, Ma0a1, Ma1a2, code;


  line[0] = WHITE; // force a white pixel on line beginning

  while (a0 < PELS_PER_LINE) { // while not EOL
  code = get_1(MR_CONTROL);
  if (code == EOF)
    return(-1);
  switch (code) {
  case V0:
    vertical_code(&a0_color, &a0, prev, PELS_PER_LINE, line, 0);
    break;
  case VR1:
    vertical_code(&a0_color, &a0, prev, PELS_PER_LINE, line, 1);
    break;
  case VL1:
    vertical_code(&a0_color, &a0, prev, PELS_PER_LINE, line, -1);
    break;
  case HOR:
    Ma0a1 = find_huf_len(&a0_color);  // gpos is gloablly known
    Ma1a2 = find_huf_len(&a0_color);
    memset(line+a0, a0_color, Ma0a1);
    memset(line+a0+Ma0a1, !a0_color, Ma1a2);
    a0 += (Ma0a1+Ma1a2);
    break;
  case PASS:
    b1 = find_b1(a0_color, a0, prev, PELS_PER_LINE);
    b2 = find_next(a0_color, b1+1, prev, PELS_PER_LINE);
    memset(line+a0, a0_color, b2-a0);
    a0 = b2;
    break;
```

C:\ARIK\COMPRESS\PTNTSRC\AGEXP.C - Sun Aug 28 07:04:42 1994

```
    case VR2:
      vertical_code(&a0_color, &a0, prev, PELS_PER_LINE, line, 2);
      break;
    case VL2:
      vertical_code(&a0_color, &a0, prev, PELS_PER_LINE, line, -2);
      break;
    case VR3:
      vertical_code(&a0_color, &a0, prev, PELS_PER_LINE, line, 3);
      break;
    case VL3:
      vertical_code(&a0_color, &a0, prev, PELS_PER_LINE, line, -3);
      break;
    }
  }
  return(1);
}


find_b1(int a0_color, int a0, char *line, int length)
{
  int b1;

  if (line[a0] == a0_color)
    b1 = find_next(!a0_color, a0+1, line, length);
  else {
    b1 = find_next(a0_color, a0+1, line, length);
    b1 = find_next(!a0_color, b1+1, line, length);
  }

  return(b1);
}

/* Builds partial rasterized line according to MR codes */
void vertical_code(int *a0_color, int *a0, char *prev, int length, char *curr, int offset)
{
  int a1, b1;

  b1 = find_b1(*a0_color, *a0, prev, PELS_PER_LINE);
  //printf(" (b1=%d)\n", b1);
  a1 = b1 + offset;
  //printf("MEMSET (vert): %d, %d, %d\t\t(a1=%d, a0=%d)\n", *a0, *a0_color, a1-*a0, a1, *a0);
  memset(curr+*a0, *a0_color, a1-*a0);
  *a0_color = !(*a0_color);
  *a0 = a1;
}


find_huf_len(int *a0_color)
{
  int len;

  len = BW_SYMBOLS - 1- get_1(*a0_color);

  if (len > 63)
    len = (len - 63) * 64;

  if (len < 64) {
```

C:\ARIK\COMPRESS\PTNTSRC\AGEXP.C - Sun Aug 28 07:04:42 1994

```
    *a0_color = !*a0_color;
    return(len);
  } else
    return(len + find_huf_len(a0_color));
}

find_next(int color, int pos, char *line, int len)
{
  int i;
  char *ptr;

  if (pos > len-1)
  return(len);

  if ( (ptr = memchr(line+pos, color, len-pos)) == NULL)
    return len;
  else
    return (ptr-line);
}

BigErr(int n, char *s)  // too many bits in strip.
{
  printf("Err %d - %s", n, s);
  exit(9);
}


static int *count, prev, prev1;
/**** Arithmetic decoder staff *****/
init_get_10
{
  count = malloc( sizeof(int) * 3); // mr + b&w
  memset(count, 0, sizeof(int) * 3);
  prev = prev1 = 0;
}

/**** Arithmetic decoder staff *****/
close_get_10
{
  free(count);
  close_input_bitstream0;
}

/***** gets one symbol from the arithmetic coder */
get_1(int mode)
{
  SYMBOL s;
  int c;

  get_symbol_scale( &s, mode, prev, prev1 );
  count(mode) = get_current_count( &s );
  c = convert_symbol_to_int( count(mode), &s );

  if (mode == MR_CONTROL) {
    prev1 = prev;
    prev = c;
```

C:\ARIK\COMPRESS\PTNTSRC\AGEXP.C - Sun Aug 28 07:04:42 1994

```
  }

  remove_symbol_from_stream( &s );

  if ( c != EOF )
    update_model( c );
  return(c);
}


static pack_bytes, byte;

/*** routines for packing bytes to bits (for output) ***/
 pack8(unsigned char *line, unsigned char *buf)
{
  int i=0, j, k, color, new_pos, pos=0, n, bits, count=0;

  pack_bytes = 0;
  byte = 0;
  color = line[0];
  while ((new_pos = find_next(color, pos, line, PELS_PER_LINE)) != PELS_PER_LINE) {
    pack_n_bits(color, new_pos - pos, &count, buf);
    pos = new_pos;
    color = !color;
  }
  pack_n_bits(color, PELS_PER_LINE - pos, &count, buf);
}

pack_n_bits(int color, int n, int *count, char *buf)
{
  int bits;
  static b_table[] = {0,1,3,7,15,31,63,127,255};

  while ( (*count+n) > 8) {
  if (*count != 0) {
    bits = 8 - *count;
    byte = (byte << bits);
    if (color)
      byte += ( (color << bits) - 1 );
    buf[pack_bytes] = byte;
    pack_bytes++;
    n-=bits;
    *count = 0;
  } else {
    if (color)
       byte = 255;
    else
    byte = 0;
    buf[pack_bytes] = byte;
    pack_bytes++;
    n-=8;
  }
  }

  byte = (byte << n);
  if (color)
    byte += b_table[n];
```

C:\ARIK\COMPRESS\PTNTSRC\AGEXP.C - Sun Aug 28 07:04:42 1994

```
  (*count) += n;

  if (*count == 8) {
     buf[pack_bytes] = byte;
     pack_bytes++;
     *count = 0;
  }
}
```